Graph Databases for Smart Cities

Martin Brugnara*, Matteo Lissandrini†, Yannis Velegrakis*
*University of Trento, †Aalborg University
mb@disi.unitn.eu, matteo@cs.aau.dk, velgias@disi.unitn.eu

***Abstract*—Smart Cities generate continuously a huge amounts of graph data. The collection, processing, and analysis of such data is crucial for the prosperity and progress of the cities. Many specialized solutions have been developed for dealing with graph data. Among them are Graph Database Systems that offer storage and querying for graph data. In this paper we present a detailed and curated selection of such systems that offer a standard interaction method via the Apache TinkerPop's Gremlin protocol and report the summary of our detailed evaluation of such systems. We also analyze some scenarios in the domain of Smart Cities and, based on our findings, suggest the most appropriate system to use in each use case.**

I. Introduction

City management has, for quite some time, relied on many different information sources like economic analysis, polls, and ad-hock studies, to monitor the evolution of the city and identify possible areas of improvement. Today, thanks to cheap IoT devices, improved network connectivity, and the advancements in data processing and data analysis, cities are equipped with tons of different monitoring devices that provide real time feedback on everything is happening. These new flood information allows administrations to proactively and promptly react and intervene to optimize ongoing processes or stop unwanted situation. The volume, variety, and velocity of the data collected by Smart Cities is so extreme are characterized as Big Data.

While graphs have been in use for almost 400 years, in recent years we witnessed an increased interest in systems that manage and process them. Such systems have become increasingly important for a wide range of applications [1], [2] and domains, including biological data [3], knowledge graphs [4], and social networks [5]. In the past graphs have been stored in relational databases [6] and in Resource Description Framework (RDF) stores [7]. While these solutions may have served well in conjunction with ad-hoc applications, they do not typically offer common operations on graphs like shortest-path, triangle discovery, or path filtering. Even though a subset of such functions may be found implemented via custom procedures, they tend to have poor performance due to the back stores that are not designed for these applications [8], [9]. To address these shortcomings a new class of databases has been developed: the Graph Databases.

Graph databases build on the foundations of network and relational databases to provide graph oriented functionalities, to implement specialized optimizations and offer indexes. A number of such systems have been recently developed. Even though they share a similar set of features, they are built on very different technologies and techniques. Since they usually provide different interaction methods, a recent effort proposes a standardized access method: the Apache TinkerPop project [10], [11].

In this paper we provide a survey of the newborn database systems that support the TinkerPop project's standardized access method Gremlin. For each system we provide a description of its capabilities, an illustration of its internal functions, and insights on its theoretical strong and weak spots. We then report the result of our empirical comparison [12] of these systems. We conclude by analyzing some common scenarios in the context of Smart Cities and suggest the systems that best fit for each use case.

II. Systems

In this section we present a selected set of graph database systems. The selection criteria included support for a common access method. Tinkerpop [11], an open source, vendor-agnostic, graph computing framework, is currently the only common interface in most graph databases. TinkerPop-enabled systems are able to process a common query language: the Gremlin language. Thus, we chose systems that support some version of it through officially recognized implementations. Furthermore, we consider systems with a licence that permits the publication of experimental evaluations, and also those that were made available to us to run on our server without any fee. Table I summarizes the characteristics of the systems we consider in our study. Among others, we show the query languages that these systems support (other than Gremlin).

There are two ways to implement a graph database. One is to build it from scratch (*native* databases) and the other to achieve the required functionalities through other existing systems (*hybrid* databases). In both cases the two challenges to solve are how to store the data and how to traverse these stored structures.

A. Native System Architectures

For data storage, a common design principle is to separate information about the graph structure (nodes and edges) from other they may have, e.g., attribute values, to speed-up traversal operations.

Neo4J :: Neo4j [13] is a database system implemented in Java and distributed under both an open source and commercial licence. It provides its own unique language called Cypher, and supports also Gremlin, and native Java API. It employs a custom disk-based native storage engine where nodes, relationships, and properties are stored separately on disk. Dynamic pointer compression expands the available address space as needed, allowing the storage of graphs of any

TABLE I
FEATURES AND CHARACTERISTICS OF THE TESTED SYSTEMS

System	Type	Storage	Edge Traversal	Gremlin	Query Execution	Access	Languages
ArangoDB (2.8)	Hybrid (Document)	Serialized JSON	Hash Index	v2.6	AQL, Non-optimized	REST (V8 Server)	AQL, Javascript
BlazeGraph (2.1.4)	Hybrid (RDF)	RDF statements	B+Tree	v3.2	Programming API, Non-optimized	embedded, REST	Java, SPARQL
Neo4J (1.9, 3.0)	Native	Linked Fixed-Size records	Direct Pointer	v2.6 / v3.2	Programming API, Non-optimized	embedded, WebSocket, REST	Java, Cypher,
OrientDB (2.2)	Native	Linked Records	2-hop Pointer	v2.6	Mixed, Mixed	embedded, WebSocket,REST	Java, SQL-like
Sparksee (5.1)	Native	Indexed Bitmaps	B+Tree/Bitmap	v2.6	Programming API, Non-optimized	embedded	Java, C++,Python, .NET
SQLG (1.2) / Postgres (9.6)	Hybrid (Relational)	Tables	Table Join	v3.2	SQL, Optimized(*)	embedded (JDBC)	Java
Titan (0.5, 1.0)	Hybrid (Columnar)	Vertex-Indexed Adjacency List	Row-Key Index	v2.6 / v3.0	Programming API, Optimized	embedded, REST	Java

size in its latest version. Full ACID transactions are supported through a write-ahead log. A lock manager applies locks on the database objects that are altered during the transaction.

Neo4j has in place a mechanism for fast access to nodes and edges that is based on IDs. The IDs are basically offsets in one of the store files. Hence, upon the deletion of nodes, the IDs can be reclaimed for new objects. It also supports *schema indexes* on nodes, labels and property values. Moreover, it supports full text indexes that are enabled by an external indexing engine (Apache Lucene [14]), which also allows nodes and edges to be viewed and indexed as "key:value" pairs. Other Neo4J features include replication modes and federation for high-availability scenarios, causal cluster, block device support, and compiled runtime.

Neo4J has one file for node records, one file for edge records, one file for labels and types, and one file for attributes. Node and edge records contain pointers to other edges and nodes, and also to types and attributes. In Neo4J nodes and edges are stored as records of fixed size and have unique IDs that correspond to the offset of their position within the corresponding file. In this way, given the id of an edge, it is retrieved by multiplying the record size by its id, and reading bytes at that offset in the corresponding file. Moreover, being records of fixed size, each node record points only to the first edge in a doubly-linked list, and the other edges are retrieved by following such links. A similar approach is used for attributes.

OrientDB :: OrientDB [15] is a multi-model database, supporting graph, document, key/value, and object data models. It is written in Java and is available under the Apache Licence or a Commercial licence. Its multi-model features Object-Oriented concepts with a distinction for classes, documents, document fields, and links. For graph data, a node is a document, and an edge is mapped to a link. Various approaches are provided for interacting with OrientDB, from the native Java API (both document-oriented and graph-oriented), to Gremlin, and extended SQL, which is a SQL-like query language.

OrientDB features 3 storage types: (i) *plocal*, which is a persistent disk-based storage accessed by the same JVM process that runs the queries; (ii) *remote*, which is a network access to a remote storage; and (iii) *memory-based*, where all data is stored into main memory. The disk based storage (also called Paginated Local Storage) uses a page model and a disk cache. The main components on disk are files called *clusters*. A cluster is a logical portion of disk space where OrientDB stores record data, and each cluster is split into pages, so that each operation is atomic at page level.

OrientDB supports ACID transactions through a write ahead log and a *Multiversion Concurrency Control* system where the system keeps the transactions on the client RAM. This means that the size of a transaction is bounded by the JVM available memory. OrientDB also implements SB−Tree indexes (based on B-Trees), hash indexes, and Lucene full text indexes. The system can be deployed with a client-server architecture in a multi-master distributed cluster.

OrientDB stores information about nodes, edges and attributes similarly, in distinct records. Node and edge records contain pointers to other edges and nodes, and also to types and attributes. In OrientDB, differently from Neo4J, record IDs values are not linked directly to a physical position, but point to an append-only data structure, where the logical identifier is mapped to a physical position. This allows for changing the physical position of an object without changing its identifier. In both cases, given an edge, to obtain its source and destination requires constant time operations, and inspecting all edges incident on a node, hence visiting the neighbors of a node, has a cost that depends on the node degree and not on the graph size.

Sparksee :: Sparksee [16], [17], formerly known as DEX [18], is a commercial system written in C++ optimized for out-of-core operations. It provides a native API for Java, C++, Python and .NET platforms, but it does not implement any other query language apart from Gremlin.

It is specifically designed for labeled and attributed multi-graphs. Each vertex and each edge are distinguished by permanent object identifiers. The graph is then split into multiple lists of pairs and the storage of both the structure and the data is partitioned into different clusters of bitmaps for a compact representation. This data organization allows for more than 100 billion vertices and edges to be handled by a single machine. Bitmap clusters are stored in sorted tree structures that are paired with binary logic operations to speedup insertion, removal, and search operations.

Sparksee supports ACID transaction with a *N-readers* and *1−writer* model, enabling multiple read transactions with each write transaction being executed exclusively. Both search and unique indexes are supported for node and edge attributes. In addition a specific neighbor index can also be defined to improve certain traversal operations. It provides horizontal scaling, enabling several slave databases to work as replicas of a single master instance.

Sparksee decomposes data into separate data-structures: one structure for objects, which refers to both nodes and edges, two for relationships which describe which nodes and edges

are linked to each other, and a data-structure for each attribute name. Each of these data-structures is in turn composed by a map from keys to values, and a bitmap for each value [17]. In each data-structure objects are identified by IDs generate sequentially, and each ID is linked as key trough the map to one single value. Also, each value links to a bitmap, where each bit corresponds to an object ID, and the bit is set if that object has that value. Given a label, one can scan the corresponding bitmap to identify which edges share the same label. Furthermore, each bitmap identifies all edges incident to a node. For the attributes a similar mechanism is used. The main advantage of this organization is that many operations become bitwise operations on bitmaps, although operations like edge traversals have no constant time guarantees.

B. Hybrid System Architectures

ArangoDB :: ArangoDB [19] is a multi-model database. This means that it can work as a document store, a key/value store and a graph database, all at the same time. With this model, objects like nodes, edges or documents, are treated the same and stored into special structures called collections. Apart from Gremlin, it supports its own query language, called AQL, *ArangoDB Query Language*, which is an SQL like dialect that supports multiple data models with single document operations, graph traversals, joins, and transactions. The core, which is open-source (Apache License), is written in C++, and is integrated with the V8 JavaScript Engine[1]. That means that it can run user-defined JavaScript code, which will be compiled to native code by V8 on the fly, while AQL primitives are written in C++ and will be executed as such. Nonetheless, the supported way of interacting with the database system is via REST API and HTTP calls, meaning that there is no direct way to embed the server within an application, and that every query will go through a TCP connection.

It supports ACID transactions by storing data modification operations in a write-ahead log, which is a sequence of append-only files containing every write operations executed on the server. While ArangoDB automatically indexes some system attributes (i.e., internal node identifiers), users can also create additional custom indexes. As a consequence, every collection (documents, nodes or edges) has a default primary index, which is an unsorted hash index on object identifiers, and, as such, it can be used neither for non-equality range queries nor for sorting. Furthermore, there exists a default edge index providing for every edge quick access to its source and destination. ArangoDB can serve multiple requests in parallel and supports horizontal scale-out with a cluster deployment using Apache Mesos [20].

ArangoDB is based on a document store. Each document is represented as a self contained JSON object (serialized in a compressed binary format). To implement the graph model, ArangoDB materialize JSON objects for each node and edge. Each object contains links to the other objects to which it is connected, e.g., a node lists all the IDs of incident edges. A specialized hash index is in place, in order to retrieve the source and destination nodes for an edge, this speed-ups many traversals.

[1]chromium.googlesource.com/v8/v8.git

BlazeGraph :: Blazegraph [21] is open-source and available under GPLv2 or under a commercial licence. It is an RDF-oriented graph database entirely written in Java. Other than Gremlin, it supports SPARQL 1.1, storage and querying of reified statements, and graph analytics.

Storage is provided through a journal file with support for index management against a single backing store, which scales up to ∼50B triples or quads on a single machine. Full text indexing and search facility are built using a key-range partitioned distributed B+Tree architecture. The database can also be deployed in different modes of replication or distribution. One of them is the federated option that implements a scale-out architecture, using dynamically partitioned indexes to distribute the data across a cluster. While updates on the journal and the replication cluster are ACID, updates on the federation are *shard-wise ACID*. Blazegraph uses Multi-Version Concurrency Control (MVCC) for transactions. Transactions are validated upon commit using a unique timestamp for each commit point and transaction. If there is a write-write conflict the transaction aborts. It can operate as an embedded database, or in a client-server architecture using a REST API and a SPARQL end-point.

BlazeGraph is an RDF database and stores all information into Subject-Predicate-Object (SPO) triples. Each statement is indexed three times by changing the order of the values in each triple, i.e., a B+Tree is built for each one of SPO, POS, OSP. BlazeGraph stores attributes for edges as reified statements, i.e., each edge can assume the role of a subject in a statement. Hence, traversing the structure of the graph may require more than one accesses to the corresponding B+Tree.

Sqlg/Postgresql :: Sqlg [22] is an implementation of Apache TinkerPop on a relational DBMS. Postgresql [23] is one among those RDBMS supported, and the one we chose for our experiments. Sqlg provides Java API to the gremlin language, and the underlying implementation maps graph semantics to that of the RDBMS. It is possible to also send standard SQL queries directly to the back-end relational database, although this is not often convenient. For graph data, a vertex label is modeled by a table, containing all vertices with that label, and all the vertex's properties. An edge label is modeled as a many-to-many join-table between the vertices, Similarly to vertices, edge labels are mapped to tables. containing the vertex ID of the two edge endpoints alongside the edge properties. Indexes, transactions and parallelization are inherited from the underlying database system.

In **Sqlg** the graph structure consists of one table for each edge type, and one table for each node type. Each node and edge is identified by a unique ID, and connections between nodes and edges are retrieved through joins. The limitation of this approach is that unions and joins are required even for retrieving the incident edges of a node.

Titan :: Titan [24][2] is available under the Apache License. The main part of the system handles data modeling, and query execution, while the data-persistence is handled by a third-party storage and indexing engine to be plugged into it. For storage, it can support an in-memory storage engine (not intended for production use), Cassandra [25], HBase [26], and BerkeleyDB [27]. To store the graph data, Titan adopts the adjacency list format, where each vertex is stored alongside the list of incident edges. In addition, each vertex property is an entry in the vertex record. Titan adopts Gremlin as its only query language, and Java as the only compatible programming interface. No ACID transactions are supported in general, but are left to the storage layer that is used. Among the three available storage backends only Berkeley DB supports them. Cassandra and HBase provide no serializable isolation, and no multi-row atomic writes.

Titan supports two types of indexes: *graph centric* and *vertex centric*. A graph index is a global structure over the entire graph that facilitates efficient retrieval of vertices or edges by their properties. It supports equality, range, and full-text search. A Vertex-centric index, on the other hand, is local to each specific vertex, and is created based on the label of the adjacent edges and on the properties of the vertex. It is used to speed up edge traversal and filtering, and supports only equality and range search. For more complex indexing external engines like Apache Lucene or ElasticSearch [28] can be used. Due to the ability of Cassandra and HBase to work on a cluster, Titan can also support the same level of parallelization in storage and processing.

Titan represents the graph as a collection of adjacency lists. With this model the system generates a row for each node, and then one column for each node attribute and each edge. Hence, for each edge traversal, it needs to access the node (row) ID index first.

C. Differences in Query Processing

All the systems we considered support Gremlin. A Gremlin query is a series of operations. Consider, for instance, the following query `g.V.filter{it.inE.count()>=k}`, which selects nodes with at least k incoming edges. It first filters nodes (`g.V.filter{...}`) and then the incoming edges are counted (`it.inE.count()`) for every node in the output of the filter.

In **ArangoDB** each step is converted into an AQL query and sent to the server for execution, so the above Gremlin query will be executed as a series of two independent AQL queries implementing its two parts. ArangoDB does not provide any overall optimization of these parts. Note that Gremlin is a turing-complete language and can describe complex operations that declarative languages, like AQL or Cypher, may not be able to express in one query. The only other query system that translates all operations to a declarative query language is **Sqlg**. Where possible, the system tries to conflate operators in a single query, which is some form of query optimization. All the other systems translate Gremlin queries directly into a sequence of low-level operators with direct access to their programming API, evaluate every operator, and pass the result to the next in the sequence. In **OrientDB**, in particular, some consequent operators may get translated into queries and then the processed with the programming API, resulting in some form of optimization for a part of the query. **Titan**, which has Gremlin as the only supported query language, features also some optimization during query processing.

[2]At the time of writing, Janus Graph has just started as the successor of Titan, yet here we discuss the last version of Titan.

III. System Selection

In a recent study we have extensively evaluated these systems [12]. The evaluation was designed accordingly to the micro-benchmark philosophy. The performance of all possible elementary operation on graph data have been tested by repeated experiments in a controlled environment. The results are summarized in Table II. Each column represents an operation, each row represents a graph database system. Operations are grouped by type: load (L), creation (C), read (R), update (U), delete (D), traversal (T). Each cell contains a score computed relatively to the other solutions. It may either be positive ✓, negative ⚠, or neutral (blank).

It is easy to see from that table that there is no clear winner. The choice of the system to be used in each specific situation will depend on the requirements of the considered use case. To make this point more clear, let us consider some use case scenarios that are commonly found in Smart Cities.

TABLE II
Evaluation Summary

	L	C	R			U	D		T				
	Load	Insertions	Graph Statistics	Search by Property\Label	Search by Id	Updates	Delete Node	Other Deletions	Neighbors	Node Edge-Labels	Degree Filter	BFS	Shortest Path
ArangoDB		✓	⚠	⚠		✓	✓	✓			⚠	⚠	⚠
BlazeGraph	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠
Neo4J (v.1.9)	✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓
Neo4J (v.3.0)	✓		✓						✓	✓	✓	✓	✓
OrientDB		✓			✓	✓		✓	✓	✓		✓	
Sparksee	✓	✓	✓		✓	✓		✓					⚠
Titan (v.0.5)	⚠		⚠	⚠									
Titan (v.1.0)	⚠		⚠	⚠									⚠
Sqlg				✓		✓		✓	⚠	⚠	⚠	⚠	⚠

1) Monitoring :: Smart Cities are well known for their abundance of sensors. They collect data about almost any possible aspect of the city, among them there are those dedicated to the air analysis. These in particular, are used to *monitor the quality of the breathable air and the pollution levels*; too low values in the first or too high in the latter would require an immediate response by the administration to guarantee the citizen safety. A frequently used temporary remedy consists in traffic bans. This solution can mitigate and sometimes solve the problem, but it requires up-to-date and near-real-time information. This requires efficiently filtering the measurements associated with each sensor. In a graph database this would entail *searching by property*. The solution that would best fit this scenario is **Sqlg**.

2) Zone Planning :: An example of an antipodal scenarios that requires instead analyzing historical data is that of *zone planning*. By evaluating the neighborhood of a zone of interest on different dimensions, like noise pollution, air quality, air quality and connectivity, the administration can better design the city expansion. The crucial operation here consists in collecting and elaborating for each zone of interest (set of node) the data from the nearest neighbors. System that distinguish themselves for their performance in traversing the *neighbors* are **Neo4j** and **OrientDB**.

3) Dynamic Traffic Rerouting :: A scenario were old data and live data mixes in Smart cities is *dynamic traffic rerouting*. We all experienced long queues and traffic jams in the peaks hours, but now, thanks to the historical data and the live feedback from the streets, this can change. Live traffic patterns can be matched with historical data to predict possible traffic jams and smart road signs can be updated to enable different routes and manage the stream of vehicles. Clearly the same strategy may be applied whenever car crashes should occur. This task requires to efficiently find new optimal path to evacuate the cars from the current location and directing them to their most probable final locations – this would likely addressed per destination zones and not singles addresses. The problem of computing such paths is known as *shortest path finding*, and the system that may better help is **Neo4j**.

4) Isochrone Maps :: Another scenario that relies on stable data and live updates consists in providing a service that generates on demand isochrone maps. Such service may, for example, empower visitors to plan their exploration, transportation company to optimize their deliveries, and the administration itself to plan for future transport development. Isochrone maps are generated by computing a *weighted breath first search* (WBFS), with the weight set as the time to travel through a street (edge). Also in this case, **Neo4j** is a suitable solution alongside **OrientDB**.

IV. Conclusion

We have presented seven Graph Database Systems. For each system, we have provided a description of its capabilities, an illustration of its internal functions, and insights on its theoretical strong and weak spots. With such information alongside the insight provided from an experimental comparison we have presented elsewhere [12], we have analyzed four scenarios in the context of Smart Cities and suggested the best solution for each use case.

V. Acknowledgment

This work has been supported by the IEEE Smart Cities Initiative, Student Grant Program.

References

[1] M. Lissandrini, D. Mottin, T. Palpanas, D. Papadimitriou, and Y. Velegrakis, "Unleashing the power of information graphs," *SIGMOD Rec.*, vol. 43, no. 4, pp. 21–26, Feb. 2015. [Online]. Available: http://doi.acm.org/10.1145/2737817.2737822

[2] E. Bullmore and O. Sporns, "Complex brain networks: graph theoretical analysis of structural and functional systems," *Nature Reviews Neuroscience*, vol. 10, no. 3, p. 186, 2009.

[3] D. Bu, Y. Zhao, L. Cai, H. Xue, X. Zhu, H. Lu, J. Zhang, S. Sun, L. Ling, N. Zhang *et al.*, "Topological structure analysis of the protein–protein interaction network in budding yeast," *Nucleic acids research*, vol. 31, no. 9, pp. 2443–2450, 2003.

[4] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: a core of semantic knowledge," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 697–706.

[5] O. Goonetilleke, S. Sathe, T. Sellis, and X. Zhang, "Microblogging queries on graph databases: An introspection," in *GRADES*. New York, NY, USA: ACM, 2015, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2764947.2764952

[6] R. N. Goldberg and G. A. Jirak, "Relational database management system and method for storing, retrieving and modifying directed graph data structures," Apr. 6 1993, uS Patent 5,201,046.

[7] A. Harth, J. Umbrich, A. Hogan, and S. Decker, "YARS2: A federated repository for querying graph structured data from the web," in *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007.*, ser. Lecture Notes in Computer Science, K. Aberer, K. Choi, N. F. Noy, D. Allemang, K. Lee, L. J. B. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber, and P. Cudré-Mauroux, Eds., vol. 4825. Springer, 2007, pp. 211–224. [Online]. Available: https://doi.org/10.1007/978-3-540-76298-0_16

[8] R. Angles and C. Gutiérrez, "Survey of graph database models," *ACM Comput. Surv.*, vol. 40, no. 1, pp. 1:1–1:39, 2008. [Online]. Available: https://doi.org/10.1145/1322432.1322433

[9] R. Angles, "A comparison of current graph database models," in *Workshops Proceedings of the IEEE 28th International Conference on Data Engineering, ICDE 2012, Arlington, VA, USA, April 1-5, 2012*, A. Kementsietsidis and M. A. V. Salles, Eds. IEEE Computer Society, 2012, pp. 171–177. [Online]. Available: https://doi.org/10.1109/ICDEW.2012.31

[10] M. A. Rodriguez, "The gremlin graph traversal machine and language (invited talk)," in *DBPL*, 2015, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/2815072.2815073

[11] "Apache Tinkerpop," http://tinkerpop.apache.org/.

[12] M. Lissandrini, M. Brugnara, and Y. Velegrakis, "Beyond Macrobenchmarks: Microbenchmark-based Graph Database Evaluation," in *VLDB'19, Proceedings of 45th International Conference on Very Large Data Bases, August 26-30, 2019, Los Angeles, California, USA*, 2019.

[13] "Neo4j," http://neo4j.com.

[14] "Apache Lucene," http://lucene.apache.org.

[15] "Orientdb," http://orientdb.com/orientdb/.

[16] "Sparsity Technologies, Sparksee," http://www.sparsity-technologies.com/.

[17] N. Martínez-Bazan, M. A. Águila Lorente, V. Muntés-Mulero, D. Dominguez-Sal, S. Gómez-Villamor, and J.-L. Larriba-Pey, "Efficient graph management based on bitmap indices," in *Proceedings of the 16th International Database Engineering & Applications Sysmposium*, ser. IDEAS '12. New York, NY, USA: ACM, 2012, pp. 110–119. [Online]. Available: http://doi.acm.org/10.1145/2351476.2351489

[18] N. Martinez-Bazan, S. Gomez-Villamor, and F. Escale-Claveras, "Dex: A high-performance graph database management system," in *ICDEW*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 124–127. [Online]. Available: http://dx.doi.org/10.1109/ICDEW.2011.5767616

[19] "Arangodb," https://www.arangodb.com/.

[20] "Apache Mesos," http://mesos.apache.org.

[21] "Systap llc., blazegraph," https://www.blazegraph.com/.

[22] P. Martin, "Sqlg," http://www.sqlg.org/.

[23] E. Prud'Hommeaux, A. Seaborne *et al.*, "Sparql query language for rdf," *W3C recommendation*, vol. 15, 2008.

[24] "Thinkaurelius, Titan," http://titan.thinkaurelius.com/.

[25] "Apache Cassandra," http://cassandra.apache.org.

[26] "Apache Hbase," http://hbase.apache.org.

[27] "Oracle, BerkeleyDB," http://www.oracle.com/technetwork/products/berkeleydb.

[28] "Elasticsearch," http://www.elastic.co/products/elasticsearch.